CS513 Computer Networks
Report of Class Project: A Client-Server Chat Program

Dufeiyang Xu
Email: dxu4@wpi.edu
Student Number: 159806200

***Abstract*—In this project, An single-server-multi-client chat system based on TCP connection is developed and tested. It supports client-to-client chat and user login/logout event broadcasting. Both server and client are written in C++ for Linux OS, in which many advanced techniques and libraries are used, such as epoll for the server and gtk3 for the client GUI. In addition, several test tools are written for mocking various connection status between clients and the server to show their stability. All of the test results are included in this report.**

I. Project Description

This project contains two executable files, ”chatClient” and ”chatServer”, as the client end and server end of a single-server-multi-client chat application. The functions provided in the client include customize the nickname, dynamically updating online user list and whispering to other clients. The client has a GUI using gtk3 library, making it beautiful, convenient and consistent with the appearance theme of the OS desktop.

The GUI screenshots are shown in Fig. 1 and Fig. 2. In the main window, the current online users will be listed in the left panel, where you can choose who to talk to. When the user list is updated or someone send a message to you, there will be a notification appearing in the left bottom ”log” panel. The widgets on the window are updated without changing the current ”focused” or ”activated” widget, so the updates will not interrupt user input.

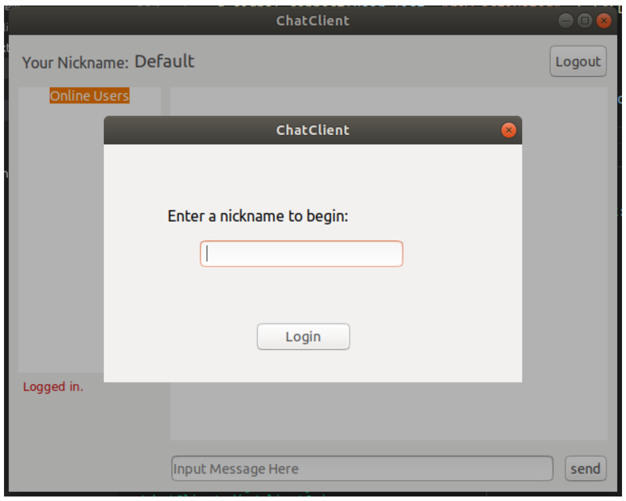
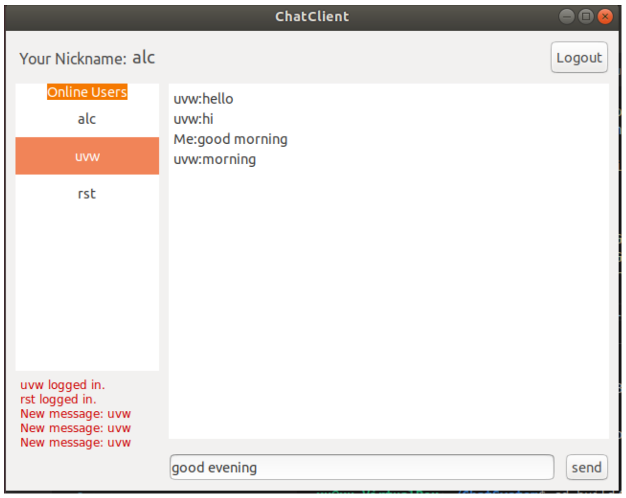


Fig. 1. Client Login Window

The server listens to the port and handles the connections of all clients. It writes log messages when user login and logout, and when the TCP connections are established and closed. The server does not has a GUI or any active functions except a ”exit” command sent by stdin. On ”exit”, the server will close all the file descriptors and streams and quit.



uvw:hello
uvw:hi
Me:good morning
uvw:morning

uvw logged in.
rst logged in.
New message: uvw
New message: uvw
New message: uvw

good evening

Fig. 2. Client Main Window

II. Detailed Design

A. Compiling Method

To compile the project from the source code, you need to install ”libgtk-3-dev” and ”CMake” as dependencies. A typical command to install these packages on Ubuntu is:

```
sudo apt-get install libgtk-3-dev cmake
```

GTK is the gui library and CMake is used as the build tool. The typical commands to build the project using CMake is:

```
mkdir build
cd build
cmake ..
cmake --build .
```

Then you will find ”chatClient” and ”chatServer” in the folder ”build”.

B. Custom Message Definition

Because we need to transfer various information between clients and the server. So it is important to define the message format clearly, shown as below:

Send by Client:

```
LOGIN:<MYNAME>;
```

```
SEND:<TONAME>:<MSG>;
LOGOUT;
```

Send by Server:

```
LOGIN_OK:<NAME0>:<NAME1>:...;
// <NAMEn> are online users,
// including the name sent by LOGIN.

LOGIN_FAILED_NICKNAME;
USER_ADD:<NAME>;
USER_DELETE:<NAME>;
RECV:<FROMNAME>:<MSG>;
SEND_FAILED:<TONAME>;
```

In addition, all the colons and semicolons in each data column are escaped with ”\:” and ”\;” when transmitted via TCP to avoid misinterpreting.

C. Server

The server is designed to handle information from three kinds of sources–the listener socket, clients sockets and stdin(used as a control method). We can notice a fact that all of the three are opened as the stream file descriptor which can be handled by a single epoll wait in a single thread. Hence, in the server, I add all of the file descriptors to epoll and start a loop to continuously call epoll_wait, and perform different actions according to the type of the descriptor returned by epoll. The usage of epoll makes the server program simple, elegant and robust. It will never be a problem to handle data race between the connections of different clients.

D. Client

For the client part, the GUI and TCP connection handler are separated, and the connection handler can run independently. The handler is designed to create a new thread to continuously receive the message from the server, and use callbacks to notify external module about an event. Though the synchronization between threads are sometimes tricky with the utilization of locks and atomic values, I leave the hard part inside the handler so that the only thing external parts need to do is providing thread-safe callbacks.

Moreover, the client data are saved with locks and marked as public inside the handler. Any external part that needs data can directly access and modify them after taking the locks.

With the proper design of the connection handler, the GUI part is more like a event-to-action mapping, with all of the client data transparent to it. Just focus on the change to GUI on different events, then a GUI is done.

III. Evaluation

In this Evaluation part, I opened 3 clients and performed a series of operations from login to logout, listing below:

1) start server;
2) start 3 clients;
3) login client 1 with name ”alc”;
4) login client 2 with name ”alc”;
5) login client 2 with name ”uvw”
6) login client 3 with name ”rst”;
7) send ”hello” from ”uvw” to ”alc”;
8) send ”hi” from ”uvw” to ”alc”;
9) send ”good morning” from ”alc” to ”uvw”;
10) input ”good evening” in client 1 (”alc”);
11) send ”morning” from ”uvw” to ”alc”;
12) logout client 2 ”uvw”;
13) send ”good evening” from ”alc” to ”uvw”;
14) server exit;

I take some representative client screenshots and show them in Figure 3, 4, 5, 6, 7, and 8 during the test steps.

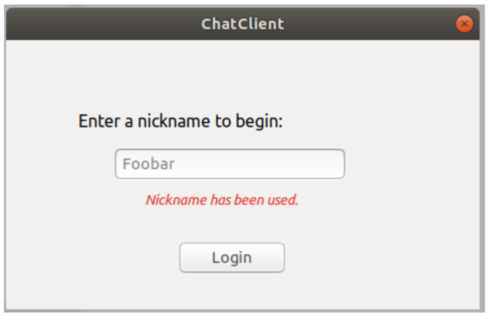
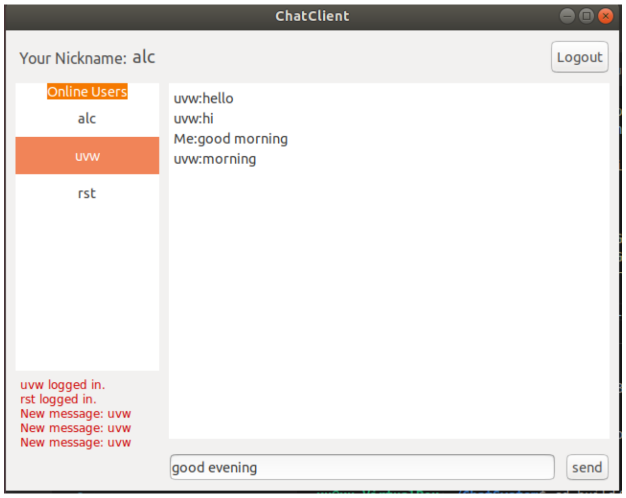


Fig. 3. Client 2 Window after Step 4



uvw:hello
uvw:hi
Me:good morning
uvw:morning

uvw logged in.
rst logged in.
New message: uvw
New message: uvw
New message: uvw

good evening

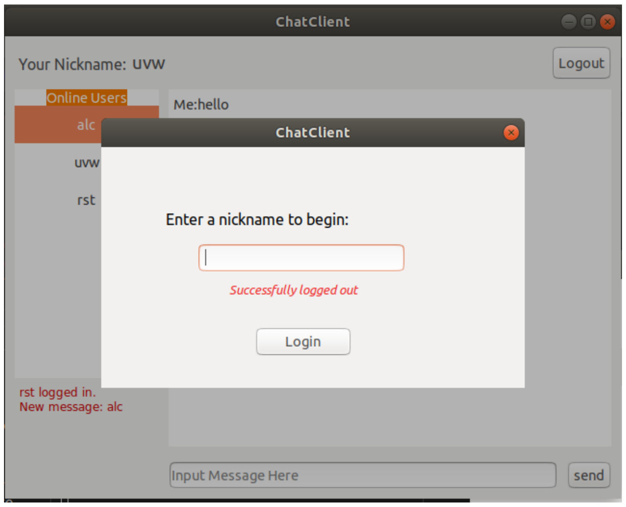
Fig. 4. Client 1 (”alc”) Window after Step 11

A screenshot of server log after Step 14 is also shown in Figure 9.

For the following part, I will use the screenshots to justify that all of the functions below work properly.

A. Normal Usage

- **Whispering to Another Client** In Figure 4, client 2 (”uvw”) send 3 messages to client 1 (”alc”), client 1 properly shows the messages in the text box and 3 log lines ”new message: uvw” in the log panel. Client 1 also shows the message that it send out, with a prefix of ”Me:”.
- **User Add/Delete Event broadcast and handling** In Figure 4, we can see there are logs of ”uvw logged in” and ”rst logged in”, and the online user list expand

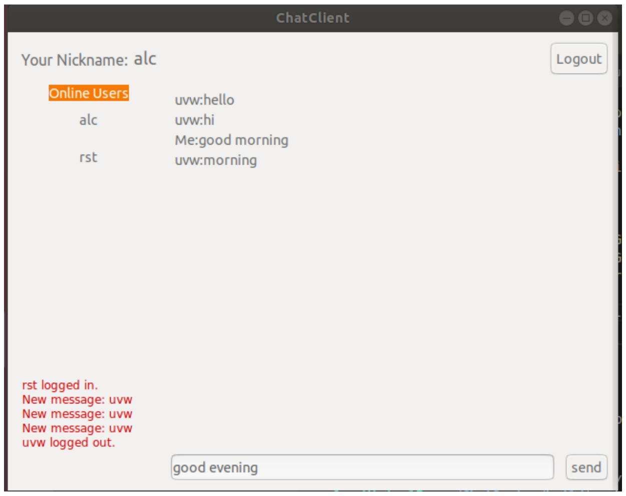


Me:hello

Successfully logged out

rst logged in.
New message: alc

Fig. 5. Client 2 (”uvw”) Window after Step 12



uvw:hello
uvw:hi
Me:good morning
uvw:morning

rst logged in.
New message: uvw
New message: uvw
New message: uvw
uvw logged out.

good evening

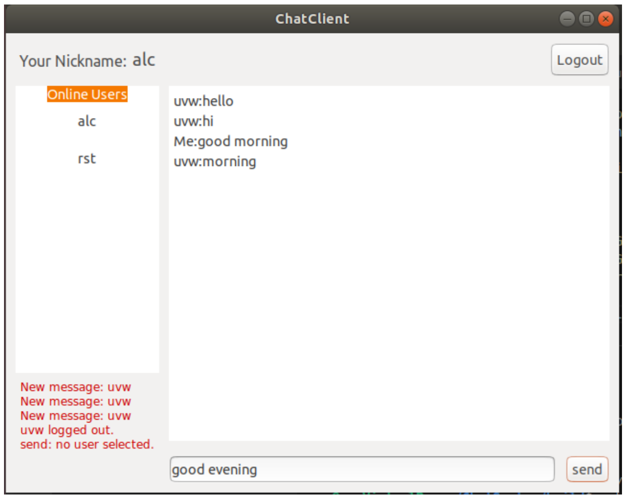
Fig. 6. Client 1(”alc”) Window after Step 12

to three items properly. Besides, In Figure 6, the client receives the ”uvw logged out” and delete the item of ”uvw” in the user list properly when client 2 logout.

- **Server log** In figure 9, the server output the correct logs when there is a client logging in or out, and when a connection is closed. It also shows the proper implementation of reading ”exit” command from the stdin which followed by a server shutdown.

B. *Error handling*

- **Nickname Conflicts** In Figure 3, I try to login client 2 with the same nickname as client 1, and the proper error message shows on the window. Also in Figure 9 we can see a ”log:login_failed_nickname:alc” indicating the server works properly.
- **Client Stable after Server Termination** In Step 14, I drop all the connections and exit the server. The client window after that is shown in Figure 8, returning to the login window with the correct error message.

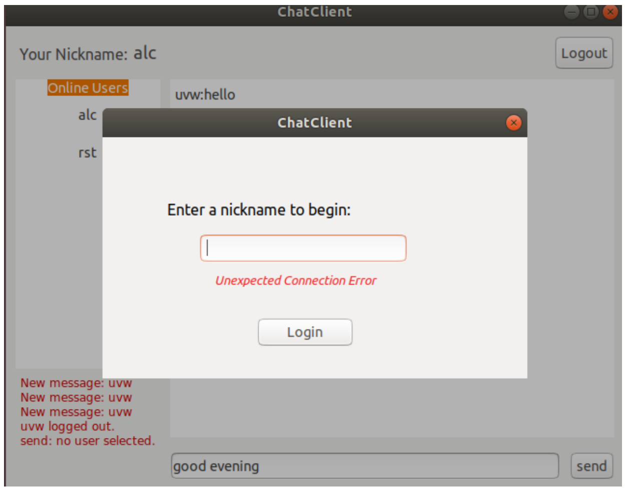


uvw:hello
uvw:hi
Me:good morning
uvw:morning

New message: uvw
New message: uvw
New message: uvw
uvw logged out.
send: no user selected.

good evening

Fig. 7. Client 1(”alc”) Window after Step 13



uvw:hello

Unexpected Connection Error

New message: uvw
New message: uvw
New message: uvw
uvw logged out.
send: no user selected.

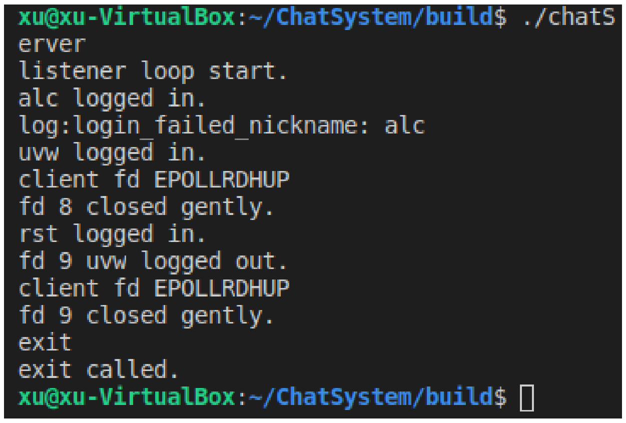
good evening

Fig. 8. Client 1 Window after Step 14

- **Server Stable after Client Termination** In Figure 9, the server is stable after fd 8 and fd 9 disconnection.
- **Whispering to a Non-Existing Client** In Figure 7, when sending message to a client that has already logged out, the client will find no online user is seleted and stop there. To test the server against this situation, I write another command line tool for forced sending a wrong message to server, shown in Figure 11. The results show that the server returns ”SEND_FAILED” message and the client call message error Callback properly.
- **Trying to Connect to Non-Existing Server** In Figure 10, I click the login button when the server is not running. The login window shows proper error message.

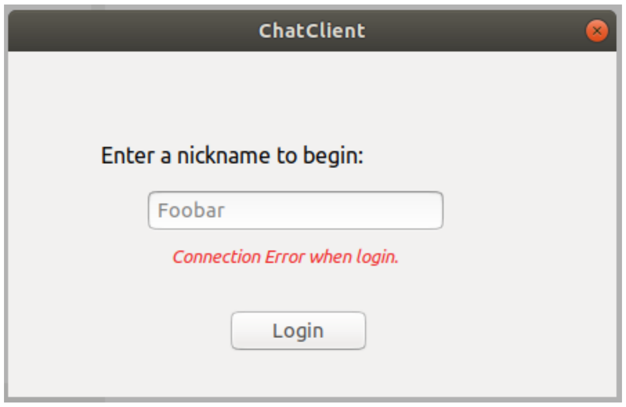
C. *TCP Connection Closing Check*

To test this, I use a ”std::cin” call to pause the client process just before the return from main(). When the process is paused, I use ”netstat | grep <my server port>” to check all of the TCP connections. The results are shown in Figure 12. The first



```
xu@xu-VirtualBox:~/ChatSystem/build$ ./chatS
erver
listener loop start.
alc logged in.
log:login_failed_nickname: alc
uvw logged in.
client fd EPOLLRDHUP
fd 8 closed gently.
rst logged in.
fd 9 uvw logged out.
client fd EPOLLRDHUP
fd 9 closed gently.
exit
exit called.
xu@xu-VirtualBox:~/ChatSystem/build$ 
```

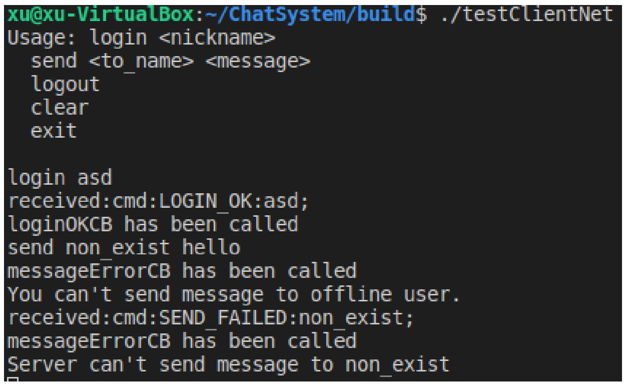
Fig. 9. Server Log after Step 14



Connection Error when login.

Fig. 10. Client Connect to Non-Existing Server

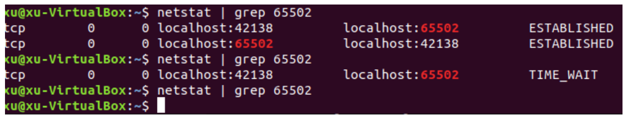
line are run when the client is logged in, showing bidirectional established TCP connection. The next two command are run when the client is paused but not terminated. From the results we find the operating system delayed the closing of the TCP to perform a TIME_WAIT but closed it eventually because we close it in the program.



```
xu@xu-VirtualBox:~/ChatSystem/build$ ./testClientNet
Usage: login <nickname>
  send <to_name> <message>
  logout
  clear
  exit

login asd
received:cmd:LOGIN_OK:asd;
loginOKCB has been called
send non_exist hello
messageErrorCB has been called
You can't send message to offline user.
received:cmd:SEND_FAILED:non_exist;
messageErrorCB has been called
Server can't send message to non_exist
```

Fig. 11. Client Forced Send A Message to Non-existing User



```
xu@xu-VirtualBox:~$ netstat | grep 65502
tcp        0      0 localhost:42138         localhost:65502         ESTABLISHED
tcp        0      0 localhost:65502         localhost:42138         ESTABLISHED
xu@xu-VirtualBox:~$ netstat | grep 65502
tcp        0      0 localhost:42138         localhost:65502         TIME_WAIT
xu@xu-VirtualBox:~$ netstat | grep 65502
xu@xu-VirtualBox:~$ 
```

Fig. 12. TCP Connection Check after Client GUI Terminate.

IV. Conclusion

In this project I develop a single-server-multi-client chat application using TCP connections. I use gtk for the client GUI and epoll for the server to handle the connections. A set of tests are performed on the implementation and demonstrate the proper behaviour and full error handling in this application.